OP2-Clang: A Source-to-Source Translator Using Clang/LLVM LibTooling

G.D. Balogh[1], G.R. Mudalige[2], I.Z. Reguly[1], S.F. Antao[3], C. Bertolli[3]

***Abstract*—Domain Specific Languages or Active Library frameworks have recently emerged as an important method for gaining performance portability, where an application can be efficiently executed on a wide range of HPC architectures without significant manual modifications. Embedded DSLs such as OP2, provides an API embedded in general purpose languages such as C/C++/Fortran. They rely on source-to-source translation and code refactorization to translate the higher-level API calls to platform specific parallel implementations. OP2 targets the solution of unstructured-mesh computations, where it can generate a variety of parallel implementations for execution on architectures such as CPUs, GPUs, distributed memory clusters and heterogeneous processors making use of a wide range of platform specific optimizations. Compiler tool-chains supporting source-to-source translation of code written in mainstream languages currently lack the capabilities to carry out such wide-ranging code transformations. Clang/LLVM's Tooling library (LibTooling) has long been touted as having such capabilities but have only demonstrated its use in simple source refactoring tasks.**

In this paper we introduce OP2-Clang, a source-to-source translator based on LibTooling, for OP2's C/C++ API, capable of generating target parallel code based on SIMD, OpenMP, CUDA and their combinations with MPI. OP2-Clang is designed to significantly reduce maintenance, particularly making it easy to be extended to generate new parallelizations and optimizations for hardware platforms. In this research, we demonstrate its capabilities including (1) the use of LibTooling's AST matchers together with a simple strategy that use parallelization templates or skeletons to significantly reduce the complexity of generating radically different and transformed target code and (2) chart the challenges and solution to generating optimized parallelizations for OpenMP, SIMD and CUDA. Results indicate that OP2-Clang produces near-identical parallel code to that of OP2's current source-to-source translator. We believe that the lessons learnt in OP2-Clang can be readily applied to developing other similar source-to-source translators, particularly for DSLs.

***Index Terms*—Source-to-source translation, Clang, LibTooling, CUDA, OpenMP, automatic parallelization, DSL, OP2, unstructured-mesh**

I. Introduction

Domain Specific Languages or Active Library frameworks provide higher-level abstraction mechanisms, using which applications can be developed by scientists and engineers. Particularly when developing numerical simulation programs for parallel high-performance computing systems, these frameworks boost both programmers' productivity and application performance by separating the concerns of programming the numerical solution from its parallel implementation. As such, they have recently emerged as an important method for gaining performance portability, where an application can be efficiently executed on a wide range of HPC architectures without significant manual modifications.

Embedded DSLs (eDSLs) such as OP2 provide a domain specific API embedded in C/C++ and Fortran [1], [2]. The API appears as calls to functions in a classical software library to the application developer. However, OP2 can then make use of extensive source-to-source translation and code refactorization to translate the higher-level API calls to platform specific parallel implementations. In the case of OP2, which provides a high-level domain specific abstraction for the solution of unstructured-mesh applications, the source-to-source translation can generate a wide range of parallel implementations for execution on architectures such as CPUs, GPUs, distributed memory clusters and emerging heterogeneous processors. The generated code makes use of parallel programming models/extensions including SIMD-vectorization, OpenMP3.0/4.0, CUDA, OpenACC and their combinations with MPI. Furthermore they implement the best platform specific optimizations, including sophisticated orchestration of parallelizations to handle data races, optimized data layout and accesses, embedding of parameters/flags that allow to tune the performance at runtime and even facilitate runtime-optimizations for gaining further performance.

Compiler tool-chains supporting source-to-source translation of code written in mainstream languages such as C/C++ or Fortran currently lack the capabilities to carry out such wide-ranging code transformations. Available tool-chains such as the ROSE compiler

*The following names are subject to trademark: LLVM™, CLANG™, NVIDIA™, P100™, CUDA™, INTEL™,

[1]G.D. Balogh and I.Z. Reguly is with the Faculty of Information Technology and Bionics, Pázmány Péter Catholic University, Hungary. balogh.gabor.daniel@itk.ppke.hu, reguly.istvan@itk.ppke.hu

[2]G.R. Mudalige is with the Department of Computer Science, University of Warwick, UK. g.mudalige@warwick.ac.uk

[3]C. Bertolli and S.F. Antao is with the IBM T.J. Watson Research Center, USA. Samuel.Antao@ibm.com, cbertol@us.ibm.com

framework [3], [4], [5] and others, have suffered from a lack of adoption by both the compilers and HPC community. This has been a major factor in limiting the wider adoption of DSLs or active libraries with a non-conventional source-to-source translator where there is a lack of a significant community of developers under open governance to maintain it. The result is often a highly complicated tool with a narrow focus, steep learning curve, inflexible for easy extension to produce different target code (e.g. new optimizations, new parallelizations for different hardware, etc.) and issues with long-term maintenance. In other words, the lack of industrial-strength tooling that can be integrated in a DevOps toolchain without any changes provokes a lack of growth in the numbers of users and developers for DSLs or active libraries, which in turn is the source of missing tools themselves, leading to a typical catch-22 or deadlock situation. The underlying motivation of the research presented in this paper is to break this deadlock by pioneering the introduction of a industrial-strength compiler toolchain, to come-up with a standard (or a methodology) that can be integrated as is in most DevOps environments.

Clang/LLVM's Tooling library (libTooling) [6] has long been touted as facilitating source-to-source translation capabilities but have only demonstrated its use in simple source refactoring and code generation tasks [7], [8] or in the transformation of a very limited subset of algorithms without hardware specific optimizations [9]. In this paper we introduce OP2-Clang [10], a source-to-source translator based on Clang's LibTooling for OP2. The broader aim is to generate platform specific parallel codes for unstructured-mesh applications written with OP2. Two options to achieve this are: (1) translating programs written with OP2's C/C++ API, to code with C++ parallelised with SIMD, OpenMP, CUDA, and their combinations with MPI etc., that can then be compiled by conventional platform specific compilers and (2) compiling programs written with OP2's C/C++ API to LLVM IR. The former case, which is the subject of this paper, follows OP2's current application development process and has been shown to deliver significant performance improvements. The latter case, which will be explored in future work, opens up the chance for application-driven low level optimizations which would otherwise be unavailable to the source-to-source OP2 solution or to the same application written as a generic C++ program. The development of OP2-Clang also aims to provide additional benefits, ones which are not easy to implement in OP2's current source-to-source translation layer written in Python. These include full syntax and semantic analysis of OP2 programs with improved user development tools to diagnose errors and correct them. Much of such capabilities come for free when using Clang/LLVM. In this research, we chart the development of OP2-Clang, outlining its design and architecture. More specifically we make the following contributions.

- We demonstrate how Clang's LibTooling can be used for source-to-source translations as required by OP2 where the target code includes wide-ranging code transformations such as SIMD-vectorization, OpenMP, CUDA and their combinations with MPI. Such a wide range of source-to-source target code generation have not been previously demonstrated either using LibTooling nor any other Clang/LLVM compiler tool.
- We present the design, including the use of LibTooling's AST matchers together with a novel strategy that use parallelization templates or skeletons to reduce the complexity of the translator. The use of skeletons allows to significantly reduce the code that needs to be generated from scratch and help modularize the translator to be easily extensible with new parallelizations.
- OP2-Clang is used for generating parallel code for two unstructured mesh applications written using the OP2 API: (1) an industrial representative Airfoil [11] CFD application and (2) a production-grade Tsunami simulation application called Volna [12], [13]. The performance of the generated parallelizations is compared to code generated by OP2's current source-to-source translator demonstrating identical performance on a number of multi-core/many-core systems.

The work described in this paper is the first step towards an application-driven full stack optimizer toolchain based on Clang/LLVM. The rest of this paper is organized as follows. In Section II we briefly introduce the motivation driving this work, including eDSLs such as OP2 and limitations of current source-to-source translation software. In Section III we chart the design and development of OP2-Clang. In Section IV we illustrate the ease of extending the tool charting the case for the SIMD-vectorization and CUDA code generator. In Section V we evaluate the performance of the generated parallel code and compare the results to the code generated by the current OP2 source-to-source translator. Section VI details conclusions and future work.

II. Background and Motivation

Source-to-source transformations and code generation have been used in many previous frameworks, particularly as a means to help application developers write programs for new hardware. Some of the earliest were motivated by the emergence of NVIDIA GPUs for scientific computations. Williams et al. [4], [5] showed that with proper annotations on the source written in

the MINT programming model [14] the ROSE compiler tool-chain [3] can be used to generate transformations to utilize GPUs. ROSE was also previously explored as a source-to-source translator tool-chain in the initial stages of the OP2 project [15]. However it proved to be hard to maintain and required a lot of coding effort to change the generated code or to adopt new parallelization models. Ueng et al. with CUDAlite [16] showed that the memory usage of existing annotated CUDA codes can be optimized with source-to-source tools based on the Phoenix compiler infrastructure. Other notable work include translators such as O_2G [17] based on the Cetus compiler framework [18] which is designed to perform source-to-source transformations based on static data dependence analysis and the hiCUDA [19] programming model which use a set of directives to transform C/C++ applications to CUDA using the front-end of the GNU C/C++ compiler. Another notable source-to-source transformation tool is Scout [20] which uses LLVM/Clang to vectorize loops using SIMD instructions at source level. The source-to-source transformation entails replacing expressions by their vectorized intrinsic counterparts. Other parallelization are not supported. To achieve this, Scout modifies the AST of the source code directly. The tools capabilities has been applied to production CFD codes at the German Aerospace Centre.

There are two main issues with the above works: (1) difficulties in extending them to generate new parallelizations or generating multiple target parallel code and (2) the underlying source-to-source translation tool relying on unmaintained software technologies due to their lack of adaptation by the community. Both these issues makes them difficult to be used in eDSLs such as OP2, which has so far relied on tools written in Python to carry out the translation of higher level API statements to their optimized platform specific versions. In fact, Python has been and continue to be used in related DSLs and active library frameworks for target code generation. These include, OPS [21], Firedrake [22], OpenSBLI [23], DeVito [24] and others. However, the tools written in Python significantly lacks the robustness of compiler frameworks such as Clang/LLVM or GNU. They do only limited syntax or semantic checking, have even limited error/bug reporting to ease the development and becomes complicated very quickly when adding different optimizations, which in turn affect its maintainability and extensibility. The present work is motivated by the need to overcome these deficiencies and aims at making use of Clang/LLVM's LibTooling for full code analysis and synthesis.

Previous work with LibTooling include [9], [25] which demonstrate its use for translation of annotated C/C++ code to CUDA using the MINT programming model. While OP2-Clang's goals are similar, in this paper we demonstrate the used of LibTooling for not only generating CUDA code, but other parallelizations such as OpenMP, SIMD and MPI. We also demonstrate how OP2-Clang can be extended with ease for new parallelizations for OP2 and apply the tool on industry representative applications.

A. *OP2*

The OP2 DSL is the second version of the Oxford Parallel Library for Unstructured mesh Solvers (OPlus), aimed at expressing and automatically parallelising unstructured mesh computations. While the first version was a classical software library, facilitating MPI parallelization, OP2 can be called an "active" library, or an embedded DSL.

The library provides an API that abstracts the solution of unstructured mesh computations. Its key components consists of (1) a mesh made up of a number of sets (such as edges and vertices), (2) connections between sets (e.g an edge is connected to two vertices), (3) data defined on sets (such as coordinates on vertices, pressure/velocity on a cell centre) and (4) Computations over a given set in the mesh. The abstraction allows the declaration of various static unstructured meshes, while extensions to it are currently planned to support mesh adaptivity.

A user initially sets up a mesh and hands all the data and metadata to the library using the OP2 API. The API appears as a classical software API embedded in C/C++ or Fortran. Any access to the data handed to OP2 can only be accessed subsequently via these API calls. Essentially, OP2 makes a private copy of the data internally and restructures its layout in any way that it sees fit to obtain best performance for the target platform. Once the mesh is set up, computations are expressed as a parallel loop over a given set, applying a "computational kernel" at each set element that uses data that is accessed either directly on the iteration set, or via at most one level of indirection. The type of access is also described - read, write, read-write or associative increment. The algorithms that can be solved by OP2 are restricted to ones in which the order of elements executed may not affect the end result to within machine precision. Additionally, users may pass mesh-invariant data to the elemental computational kernel and also carry out reductions.

This formulation of parallel loops were intentionally constructed so that it lends itself to only a handful of computational and memory access patterns: (1) directly accessed, (2) indirectly read and (3) indirectly written/incremented. Due to this high level description, OP2 is free to parallelise these loops, selecting the best implementation and optimizations for a given architecture. In other words, the abstraction allows OP2 to generate code

```
/* ----- elemental kernel function in res.h ------*/
void res(const double *edge,
         double *cell0, double *cell1 ){
  //Computations, such as:
  cell0 += *edge; *cell1 += *edge;
}

/* ---------- in the main program file -----------*/
// Declaring the mesh with OP2
// sets
op_set edges = op_decl_set(numedge, "edges");
op_set cells = op_decl_set(numcell, "cells");
// mppings -connectivity between sets
op_map edge2cell = op_decl_map(edges, cells,
                   2, etoc_mapdata,"edge2cell");
// data on sets
op_dat p_edge = op_decl_dat(edges,
                1,"double",edata,"p_edge");
op_dat p_cell = op_decl_dat(cells,
                4,"double",cdata,"p_cell");

// OP2 parallel loop declaration
op_par_loop(res,"res", edges,
  op_arg_dat(p_edge,-1,OP_ID    ,4,"double",OP_READ),
  op_arg_dat(p_cell, 0,edge2cell,4,"double",OP_INC ),
  op_arg_dat(p_cell, 1,edge2cell,4,"double",OP_INC));
```

Figure 1: Specification of an OP2 parallel loop

tailored to context. The various parallelization and performance of production applications using OP2 has been published previously in [26], [13] demonstrating near-optimal performance on a wide range of architectures including multi-core CPUs, GPUs, clusters of CPUs and GPUs. The generated parallelization makes use of an even larger range of programming models such as OpenMP, OpenMP4.0, CUDA, OpenACC, their combinations with MPI and even simultaneous heterogeneous execution.

OP2 was one of the earliest high-level abstractions frameworks to apply this development method to production applications [26]. Related frameworks for unstructured mesh applications include FeniCS [27], Firedrake [28], [29] and PyFR [30]. Other frameworks targeting a different domain include OPS [31], Devito [32], STELLA [33] (and its successor GridTools) and PSyclone [34]. In comparison, Kokkos [35] and RAJA [36] relies on C++ templates to provide a thin portability layer for parallel loops. As such the abstraction is not as high (or focused to a narrower domain) as OP2 and no code generation is carried out. OP2 is able to generate code that includes sophisticated orchestration of parallel executions, such as various colouring strategies (to handle data races), and modifications to the elemental kernel that the high-level application programmer would not have to manually implement themselves. In comparison, a Kokkos or RAJA developer would have to manually implement such changes to their code, as its abstraction is at the parallel loop level. However the advantage is that, Kokkos and RAJA are able to handle a wider range of domains.

The OP2 API was constructed to make it easy for a parsing phase to extract the relevant information about each loop that will describe which computation and memory access patterns will be used - this is required for code generation aimed at different architectures and parallelizations. Figure 1 shows the declaration of the mesh and subsequent definition of an OP2 parallel loop. In this example, the loop is over the set of `edges` in a mesh carrying out the computation per edge (which can be viewed as an elemental kernel) defined in the function `res`, accessing the data on edges `p_edge` directly and updating the data held on the two cells, `p_cell`, adjacent to an edge, indirectly via the mapping `edge2cell`. The `op_arg_dat` provides all the details of how an `op_dat`'s data is accessed in the loop. Its first argument is the `op_dat`, followed by its indirection index, `op_map` used to access the data indirectly, arity of the data in the `op_dat` and the type of the data. The final argument is the access mode of the data, read only, increment and others (such as read/write and write only not shown here).

The `op_par_loop` call contains all the necessary information about the computational loop to perform the parallelization. It is clear that due to the abstraction, the parallelization depends only on a handful of parameters such as the existence of indirectly accessed data or reductions in the loop, plus the data access modes that lends to optimizations.

The fact that only a few parameters define the parallelization means that in case of two computational loops, the generated parallel loops have the same lines of code with only small code sections with divergences. The identical chunks of code in the generated parallel loops can be considered as invariant to the transformation or boilerplate code that should be generated into every parallel implementation without change. However, given that these sections largely define the structure of the generated code, they can be viewed as an important blueprint of the target code to be generated. This leads us to the idea of using a parallel implementation (with the invariant chunks) of a dummy loop and carry-out the code generation process as a refactoring or modification of this parallel loop. In other words, use the dummy parallel loop as a skeleton (or template) and modify it to generate the required candidate computational loop. For example, Figure 2 and Figure 3 illustrates partial parallel skeletons we can extract for the generated OpenMP implementation for direct and indirect loops.

In a direct loop all iterations are independent from each other and as such the parallelization of such a loop does not have to worry about data races. However in indirect loops there is at least one `op_dat` that is accessed using and indirection, i.e via an `op_map`. Such indirections occur when the `op_dat` is not declared on the set over which the loop is iterating over. In

```
// elemental kernel function
void skeleton(double * __restrict__ d) {}

void op_par_loop_skeleton(char const *name,
                          op_set set,
                          op_arg arg0) {
  //number of arguments
  int nargs = 1;
  op_arg args[1] = {arg0};

  /*----------- Invariant code -----------------*/
  int exec_size =
      op_mpi_halo_exchanges(set, nargs, args);
  #pragma omp parallel for
  for (int n = 0; n < exec_size; n++) {
    if (n == set->core_size)
      op_mpi_wait_all(nargs, args);
  /*--------------------------------------------*/

    // set up pointers, call elemental kernel
    skeleton(&((double *)arg0.data)[2 * n]);
  }
}
```

Figure 2: Skeleton for OpenMP (excerpt) - direct kernels

which case an `op_map` that provides the connectivity information between the iteration set and the set on which the `op_dat` is declared over is used to access (read or write depending on the access mode) the data. This essentially leads to an indirect access.

With indirect loops we have to ensure that there is no two threads writing to the same data location at the same time. This is handled through the invariant code responsible for the ordering of the loop iterations in Figure. 3. In this case OP2 orchestrates the execution of iterations using a colouring scheme [37].

Note how for both direct and indirect loops the skeleton includes calls to MPI halo exchanges using `op_mpi_halo_exchanges()` to facilitate distributed memory parallelism together with OpenMP. OP2, implements distributed memory parallelization as a classical library in the back-end. As the computation requires values from neighbouring mesh elements during a distributed memory parallel run, halo exchanges are needed to carry out the computation over the mesh elements in the boundary of each MPI partition. Additionally data races over MPI is handled by redundant computation over the halo elements [38].

III. Clang LibTooling for OP2 Code Generation

The idea of modifying the target parallelization skeletons forms the basis for the design of OP2-Clang in this work. The alternative would require the full target source generation with the information given in an `op_par_loop`. The variations to be generated for each parallelization and optimization, with such a technique, would have made the code generator prohibitively laborious to develop and even more problematic to extend and maintain. The skeletons, simply allows us to reuse code and allows

```
// elemental kernel function
void skeleton(double * __restrict__ d) {}

void op_par_loop_skeleton(char const *name,
                          op_set set,
                          op_arg arg0) {
  //number of arguments
  int nargs = 1; op_arg args[1] = {arg0};
  int ninds = 1; op_arg inds[1] = {0};

  /*----------- Invariant code -----------------*/
  int set_size =
      op_mpi_halo_exchanges(set, nargs, args);
  op_plan *Plan = op_plan_get(name, set, 256, nargs,
                              args, ninds, inds);
  int block_offset = 0;
  for (int col = 0; col < Plan->ncolors; col++) {
    if (col == Plan->ncolors_core)
      op_mpi_wait_all(nargs, args);
    int nblocks = Plan->ncolblk[col];
    #pragma omp parallel for
    for(int blockIdx = 0; blockIdx<nblocks;
        blockIdx++) {
      int blockId =
          Plan->blkmap[blockIdx +block_offset];
      int nelem = Plan->nelems[blockId];
      int offset_b = Plan->offset[blockId];
      for(int n = offset_b; n<offset_b+nelem; n++) {
  /*--------------------------------------------*/
        // Prepare indirect accesses
        int map0idx =
            arg0.map_data[n * arg0.map->dim + 0];
        // set up pointers, call elemental kernel
        skeleton(&((double *)arg0.data)[2*map0idx]);
      }
    }
  }
```

Figure 3: Skeleton for OpenMP (excerpt) - indirect kernels

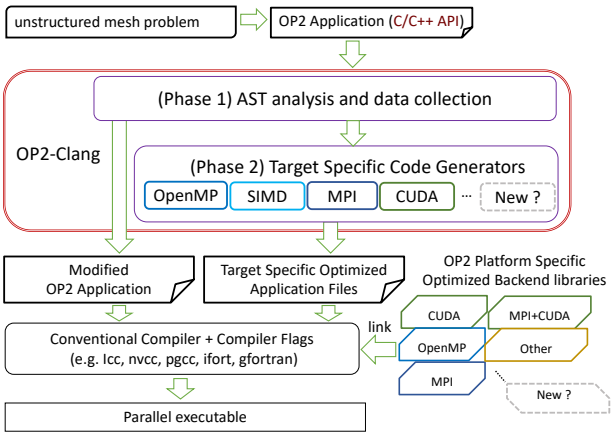

Figure 4: The high-level architecture of OP2-Clang and its place within OP2

the code generator to concentrate on the parts which need to be customized for each loop, optimization, target architecture and so on. As such we use multiple skeletons for each parallelization. In most cases one skeleton for direct and one for indirect kernels are used, given the considerable differences in direct and indirect loops as given in Figure. 2 and Figure. 3. The aim, as mentioned before, is to avoid significant structural transformation.

Clang's Tooling library (libTooling) provides a convenient API to perform a range of varying operations over

```
1   // elemental kernel function
2   void skeleton(double * __restrict__ d) {}
3
4
5
6
7
8   void op_par_loop_skeleton(char const *name,
9                             op_set set,
10                            op_arg arg0) {
11    //number of arguments
12    int nargs = 1;
13    op_arg args[1] = {arg0};
14
15    /* ------------- Invariant code ------------ */
16    int exec_size =
17        op_mpi_halo_exchanges(set, nargs, args);
18    for (int n = 0; n < exec_size; n++) {
19      if (n == set->core_size)
20        op_mpi_wait_all(nargs, args);
21    /* ----------------------------------------- */
22
23      // prepare indirect accesses
24      int map0idx = arg0.map_data[n*arg0.map->dim+0];
25
26
27      // set up pointers, call kernel
28      skeleton(&((double *)arg0.data)[2 * map0idx]);
29
30
31    }
32    // invariant code
33    ...
34  }
```

Figure 5: Parallelization skeleton for MPI (excerpt)

```
1   // elemental kernel function
2   void res(double * __restrict__ edge,
3            double * __restrict__ cell0,
4            double * __restrict__ cell1) {
5     *cell0 += *edge; *cell1 += *edge;
6   }
7
8   void op_par_loop_res(char const *name, op_set set,
9                        op_arg arg0, op_arg arg1,
10                       op_arg arg2) {
11    //number of arguments
12    int nargs = 3;
13    op_arg args[3] = {arg0, arg1, arg2};
14
15    /* ------------- Invariant code ------------ */
16    int exec_size =
17        op_mpi_halo_exchanges(set, nargs, args);
18    for (int n = 0; n < exec_size; n++) {
19      if (n == set->core_size)
20        op_mpi_wait_all(nargs, args);
21    /* ----------------------------------------- */
22
23      // prepare indirect accesses
24      int map0idx = arg1.map_data[n*arg1.map->dim+0];
25      int map1idx = arg1.map_data[n*arg1.map->dim+1];
26
27      // set up pointers, call kernel
28      res(&((double *)arg0.data)[n],
29          &((double *)arg1.data)[2 * map0idx],
30          &((double *)arg1.data)[2 * map1idx]);
31    }
32    // invariant code
33    ...
34  }
```

Figure 6: Generated MPI parallization (excerpt)

source code. As such, it's capabilities lend very well to the tasks of refactoring and source code modification of a parallelization skeleton in OP2. The starting point of source-to-source translation in OP2-Clang is to make use of Clang to parse and generate an Abstract Syntax Tree (AST) of the code with OP2 API calls. The generated AST is used to collect the required information of the sets, maps, data including their types and sizes and information in each of the parallel loops that make up the application. The second phase involves transforming the skeletons with the information for each parallel loop. The two phases of code generation and where they fit in to the overall architecture of OP2 is illustrated in Figure 4. The output of OP2-Clang will be compiled by conventional compilers, linking with OP2's platform specific libraries for a given parallelization to produce the final parallel executable. Each parallel version is generated separately.

A. OP2-Clang Application Processor

The first phase of OP2-Clang is responsible for collecting all data about the parallel loops, or kernel calls, used in the application. This step will also do semantic checks based on the types of the `op_dat`s. Particularly check whether the declared `op_dat`s match the types declared in the `op_par_loop`. However, such checks are not currently implemented, but will be very straightforward to do so given that all the information is present in the AST. The data collection and model creation happens along the OP2 API calls. As the parser builds the AST with the help of `ASTMatchers` [39], OP2-Clang keeps a record of the kernel calls, global constants and global variable declarations that are also accessible inside kernels.

During the data collection, this phase also carries out a number of modifications to the application level files that contain the OP2 API calls. Essentially replacing the `op_par_loop` calls with the function calls that implement the specific parallel implementations. Of course these implementations are yet to be generated in the second phase of code generation.

B. OP2-Clang Target Specific Code Generators

The second phase is responsible for generating the target-specific optimized parallel implementations for each of the kernels whose details are now collected within OP2-Clang. Given the information of each of the `op_par_loop`s the parallelization (currently one of OpenMP, CUDA, SIMD or MPI) we are generating code for and whether the loop is a direct or indirect loop, a target skeleton can be selected.

The target code generation then, is a matter of changing the selected skeleton at appropriate places, using the properties of the candidate `op_par_loop` for which you are going to generate a parallelization for. For example, consider the `op_par_loop` in Figure 1. This loop is an indirect loop with three arguments, one of

```
FunctionDecl op_par_loop_skeleton
...
`-CallExpr 'void'
  |-DeclRefExpr 'void (double *)' lvalue
  |   Function 'skeleton'
  `-UnaryOperator 'double *' prefix '&'
    `-ArraySubscriptExpr 'double' lvalue
      ( ... )
```

Figure 7: Elemental function call in the AST of the skeleton

```
StatementMatcher funcitonCallMatcher =
callExpr(
  callee(functionDecl(hasName("skeleton"),
                      parameterCountIs(1))),
         hasAncestor(
           functionDecl(
             hasName("op_par_loop_skeleton"))))
  .bind("function_call");
```

Figure 8: ASTMatcher to match the AST node for the elemental function call

which is a directly accessed `op_dat` and two of which are indirectly accessed `op_dats`. The loop iterates over the set of `edges` and the elemental computation kernel is given by `res`. Figure 5 shows the simplest skeleton in OP2-Clang, the skeleton for generating MPI parallelizations and Figure 6 shows the specific code that needs to be generated by changing the skeleton for the above loop. The elemental kernel function needs to be set to `res`, number of arguments set to three, while handling the indirections by using the mappings specified for those arguments and finally the elemental kernel should be called by passing in the appropriate arguments.

This type of transformations rhyme well with Clang's RefactoringTool[40], [41], which can apply replacements to the source code based on the AST. As such the process of modifying the skeleton first creates the AST of the skeleton by running it through Clang. Then the AST is searched for points of interest (i.e. points where the skeleton needs to be modified). The search is again done using `ASTMatchers` which in principal are descriptions of AST nodes of interest. For example, to set the specific elemental kernel function from `skeleton` to `res` in Figure 5, OP2-Clang needs to find the function call in the AST (part of which is given in Figure 7) using the matcher given in Figure 8. The definitions of the matchers is trivial, given that the skeleton is an input to the above process.

To formulate all such modifications to the skeleton, we create a set of matchers and run them through the `OP2RefactoringTool` which is derived from the base class `RefactoringTool` in LibTooling. When `OP2RefactoringTool` with the specific collection of matchers are run through the AST of the skeleton, the matchers find the AST nodes of interest, create a `MatchResult` object containing all the information for the given match, invokes a callback function with the

```
CallExpr *match =
    Result.Nodes.getNodeAs<CallExpr>("func_call");
SourceLocation begin = match->getLocStart();
SourceLocation end = match->getLocEnd();
SourceRange matchRange(begin, end);
string replacement =
       "res(&((double *)arg0.data)[n],"
       "&((double *)arg1.data)[2 * map0idx],"
       "&((double *)arg1.data)[2 * map1idx]);"
Replacement replacement(*Result.SourceManager,
            CharSourceRange(matchRange,false),
            replacement);
```

Figure 9: Example of creating a `Replacement` to replace the elemental function call. The `Result` object is the `MatchFinder::MatchResult` object created by the match of the ASTMatcher from Figure 8.

`MatchResult` where we can identify which matcher found a match with its keys. The identified calling matcher provides the AST node of interest and the corresponding source location. This allows to generate the specific replacement code in a `Replacement`[42] just as it is shown in Figure 9. The replacement is not immediately done, but is collected in a map. Once all the AST nodes of interest are matched and the replacement strings collected, together with the source locations, they can be applied to the source code. In this case the collection of `Replacements` are checked to ascertain if the replacements are independent from each other so that they can be applied to the source without errors. This finalizes and commits the changes to the skeleton.

As it can be seen, the process does not do large structural transformations. All the changes on the skeleton can be formulated as replacements of single lines or small source ranges. The adoption of skeletons leads to another implicit benefit. As the code generation requires the AST of the skeleton built the skeleton must be valid C/C++ code, which combined with the fact that we apply relatively small modifications on the code, implies that the generated code can also be guaranteed to be valid C/C++. Any errors can only come from small chunks of generated code which is easy for the OP2-Clang developer to debug. This is a significant benefit during the development of a new code generator. One of the key difficulties of OP2's current Python-based translator is the lack of support for such debugging tasks to ascertain that valid C/C++ code is generated.

IV. Extensibility and Modularity

The underlying aim of OP2 is to create a performance portable application written using the OP2 API. A further objective is to "future-proof" the higher-level science application where only the code generation needs to be extended to translate it to be able to execute on new hardware platforms. As such the target code generation, using OP2's Python-based translator currently supports a range

```
__constant__ intdirect_res_stride_OP2CONSTANT;
__constant__ intopDat1_res_stride_OP2CONSTANT;

__device__ void res_calc_gpu(
                const double * __restrict edge,
                double * __restrict cell0,
                double * __restrict cell1){
  cell0[0 * opDat1_res_stride_OP2CONSTANT] +=
       edge[0 * direct_res_stride_OP2CONSTANT];
  cell1[0 * opDat1_res_stride_OP2CONSTANT] +=
       edge[0 * direct_res_stride_OP2CONSTANT];
}
```

Figure 10: The modified version of the elemental function `res`, to use strided memory accesses wit SoA data layout.

```
...
// CUDA kernel function
__global__ void op_cuda_res(
            const double *__restrict arg0,
            double *ind_arg0, double *ind_arg1,
            constint *__restrict opDat0Map,
            int start, int end,
            const int * __restrict col_reord,
            int set_size) {
  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  if (tid + start >= end)
    return;
  int n = col_reord[tid + start];
  int map0idx = opDat0Map[n + set_size * 0];
  int map1idx = opDat0Map[n + set_size * 1];

  res_calc_gpu(arg0 + n,
               ind_arg0 + map0idx * 1,
               ind_arg1 + map1idx * 1);
}
...
```

Figure 11: CUDA kernel with global colouring (excerpt)

```
...
// CUDA kernel function
void op_cuda_res(const double *__restrict arg0,
                 double *ind_arg0, double *ind_arg1,
                 constint *__restrict opDat0Map,
                 int block_offset, int *blkmap,
                 int *offset, int *nelems,
                 int *ncolors, int *colors,
                 int nblocks, int set_size) {
  __shared__ int nelems2, ncolor, nelem, offset_b;
  extern __shared__char shared[];

  if (blockIdx.x >= nblocks) return;

  double arg1_l[1] = {0.0}, arg2_l[1] = {0.0};
  if (threadIdx.x == 0) {
    int blockId = blkmap[blockIdx.x + block_offset];
    nelem = nelems[blockId];
    offset_b = offset[blockId];
    ncolor = ncolors[blockId];
  }
  __synchthreads();

  int col2 = -1, n = threadIdx.x;
  if (n < nelem) {
    res_calc_gpu(arg0 + offset_b + n,
                 arg1_l, arg2_l);
    col2 = colors[n + offset_b];
  }
  for (int col = 0; col < ncolor; col++) {
    if (col2 == col) {
      int map0idx = opDat0Map[n+offset_b+set_size*0];
      int map1idx = opDat0Map[n+offset_b+set_size*1];
      ind_arg0[0 + map0idx * 1] += arg1_l[0];
      ind_arg1[0 + map1idx * 1] += arg2_l[0];
    }
    __synchthreads();
  }
}
...
```

Figure 12: CUDA kernel with hierarchical colouring (excerpt)

of parallelizations with optimizations tailored to the underlying hardware to gain near-optimal performance. The new OP2-Clang source-to-source translator will also need to support generating all these parallelizations and optimizations, but also be easy to extend to implement new parallelizations and optimizations. In this section we present further evidence of OP2-Clang's capabilities for modular and extensible code generation for a number of architectures, carrying-out different optimizations.

Since the data collection phase is independent from the target code generation OP2-Clang only needs to add a new target-specific code generator to support generating code for a new platform or to use a new parallel programming model. As we have seen, a target-specific code generator consists of (1) a parallel skeleton (usually one skeleton for implementing direct loops and one for indirect loops) (2) A list of matchers that identify AST nodes of interest and (3) a corresponding list of Replacements that specify the changes to the code.

While the skeletons already modularizes and enables reuse of code, the matchers and the corresponding `Replacements` can also be reused. Thus, when developing a new code generator we reuse existing `ASTMatchers` and `Replacements` as required. Only the matchers and `Replacements` that do not exist need to be created from scratch.

While Clang's `ASTMatchers` is extensive, there are some AST nodes that don't have specialized matchers we required. For example there were currently no matchers to match nodes representing the OpenMP constructs. We had to extended the list of available matchers with a single matcher to match the `omp parallel for` pragma in order to be able to perform our translation conveniently. We plan to work with the community to deliver these extensions back to the mainstream Clang repository. In this section we look at several other challenges that had to be overcome when supporting the various code generators for OP2.

A. CUDA

The CUDA parallelization presented a number of challenges in code generation. Again, much of the code generation using a skeleton followed a similar process to that of the OpenMP parallelization. However, the skeleton was larger and required considerably more replacements. Nevertheless, the steps taken to do the replacements were the same. For CUDA it was necessary to accurately set the device pointers and create a CUDA kernel call that encapsulate the elemental function. OP2 handles the data movement between the device and the

host in the backend with copying the data to device arrays and update host arrays if necessary, therefore this aspect doesn't affect the code generation.

The main challenge with CUDA was implementing a number of optimizations that significantly impact the performance on GPUs, unlike the previous parallelizations that utilizes CPUs. First among these is memory accesses; the memory accesses pattern of CPUs are not optimal for GPUs. For example to gain coalesced memory accesses on GPUs, OP2 can restructure the data arrays to make the neighbouring threads read data next to each other in memory. On CPUs, with large caches, it is beneficial to organize data in an Array of Structs (AoS) layout which maximizes data reuse. However, on GPUs, the threads are performing the same operation on consecutive set elements at the same time. Organizing the data such that the data needed by consecutive threads are next to each other is more beneficial. In this case we can read one cache line and use all of the data in it. This data layout is called a Structure of Arrays (SoA) layout. To use a SoA layout OP2-Clang needs to change the indexing inside the elemental function to use a strided accesses pattern. The generated elemental function for executing `res` in CUDA is illustrated in Figure 10.

Another way to improve the memory access patterns in CUDA is to modify the colouring strategy used for indirect kernels. The colouring in the OpenMP parallelization is done such that no two threads with the same colour write to the same data locations. Then iterations with the same colour can be run in parallel. Applying this strategy to CUDA, means that thread blocks need to be coloured and no two thread blocks will write to the same data. However for CUDA, previous work [43] has shown that a further level of colouring gives better performance. In this case the threads within a thread block is also coloured to avoid data races. This two level colouring is called hierarchical colouring. Hierarchical colouring has shown to considerably improve data locality and data reuse inside CUDA blocks. The difference in the CUDA kernel function between the two colouring strategies are shown in Figures 11 and 12. The variations to the code to be generated can simply be captured again with a different skeleton, in this case a skeleton that does the hierarchical colouring. However, the required Replacements, including the data layout transformations (AoS to SoA) can be reused for both one-level colouring and hierarchical colouring skeletons.

B. SIMD vectorization

A more involved code generation task is required for SIMD vectorization on CPUs. For vectorization, OP2 attempts to parallelise over the iteration set of the loop [44]. The idea is to generate code that will be automatically vectorized when compiled using a conventional

```
...
// vectorized elemental function
inline void res_vec(const double edge[*][SIMD_VEC],
                    double cell0[*][SIMD_VEC],
                    double cell1[*][SIMD_VEC], i) {
  cell0[0][i] += edge[0][i];
  cell1[0][i] += edge[0][i];
}
...

#pragma novector
for(int n=0; n<(exec_size/SIMD_VEC)*SIMD_VEC;
    n+=SIMD_VEC) {
  double arg0_p[1][VEC];
  double arg1_p[1][VEC];
  double arg2_p[1][VEC];

  //gather data to local variables
  #pragma omp simd
  for ( int i = 0; i < SIMD_VEC; i++ ) {
    arg0_p[0][i] = (ptr0)[idx0_2 + 0];
    arg1_p[0][i] = 0.0;
    arg2_p[0][i] = 0.0;
  }
  //vectorized elemental function call
  #pragma omp simd
  for ( int i = 0; i < SIMD_VEC; i++ ) {
    res_vec(arg0_p, arg1_p, arg2_p, i);
  }
  // Scatter indirect increments
  for ( int i = 0; i < SIMD_VEC; i++ ) {
    int map0idx = arg1.map_data[(n + i) *
                  arg1.map->dim + 0];
    int map1idx = arg1.map_data[(n + i) *
                  arg1.map->dim + 1];
    ((double *)arg1.data)[2 * map0idx] += arg1_p[i];
    ((double *)arg2.data)[2 * map1idx] += arg2_p[i];
  }
}
// remainder loop
for ( int n = 0; n < exec_size; n++ ) {
  int map0idx = arg1.map_data[n * arg1.map->dim + 0];
  int map1idx = arg1.map_data[n * arg1.map->dim + 1];
  res(&((double *)arg0.data)[n],
      &((double *)arg1.data)[2 * map0idx],
      &((double *)arg1.data)[2 * map1idx]);
}
...
```

Figure 13: Vectorized loop for `res` (excerpt).

C/C++ compiler such as `icpc`. Figure 13 illustrates the code that needs to be generated by OP2-Clang to achieve vectorization for our example loop `res`. There are two key difference here, compared to non-vectorizable code as in Figure 6: (1) the use of gather/scatters when indirect increments are applied and (2) the use of a modified elemental function in the vectorized loop. The first is motivated due to the multiple-iterations (equivalent to the SIMD vector length of the CPU) that are carried out simultaneously. In this case we need to be careful when indirect writes are performed. In each iteration we perform a gather of the required data to local arrays than perform the computation on the local copies and then we perform a scatter to write back the updated values. The gathers and the execution of the kernel is vectorized with `#pragma omp simd`, the scatter cannot be vectorized due to data races and so is executed serially. Finally, the remainder of the iteration set needs to be completed. Much of the code in Figure 13 can be generated as

	SIMD	OpenMP	CUDA Global (AoS)	CUDA Global (SoA)	CUDA Hierarchical (AoS)	CUDA Hierarchical (SoA)
Airfoil	363.92 s (-0.2%)	70.417 s (1.2%)	12.77 s (-0.6%)	9.58 s (-0.4%)	9.85 s (0.2%)	7.30 s (1.8%)
Volna	95.39 s (0.3%)	14.84 s (-0.2%)	3.00 s (0.5%)	2.33 s (0.2%)	2.32 s (1.2%)	1.97 s (1.1%)

Table I: Performance of Airfoil and Volna on the Intel Xeon E5-1660 CPU (for OpenMP and SIMD) and on an NVIDIA P100 GPU with OP2-Clang . CUDA results with two different colourings (global and hierarchical) and two data layouts (AoS and SoA) presented. The values in parenthesis are the percentage difference in run time compared to the sources generated with OP2's current Python-based source-to-source translator (negative values means OP2-Clang has better performance).

discussed previously. However, now we also need to modify the internals of the elemental function `res` to produce a vectorizable elemental kernel `res_vec`. The function signature needs to be changed as illustrated, which in turn requires modifications to the data accesses inside the function body (i.e. the computational kernel).

In order to perform these transformations we introduced a further layer of refactoring which parses the elemental function and transform it to the vectorized version. Since the elemental function consists of the kernel that each iteration of the loop performs, the scope of these transformations is limited. Even the indexing of the arrays are done in the generated code that calls the elemental function. To perform the necessary changes to the elemental function, the function itself is passed through Clang to obtain its AST, matchers are used to identify the AST nodes in the function signature and replace them with the correct array subscript (e.g `*edge` is changed to `edge[*][SIMD_VEC]`). Again `ASTMatchers` are used to identify AST nodes within the elemental kernel, replacing them with the variable with array subscripts (`edge[0]` is changed to `edge[0][i]`). Simple dereferences are replaced with `[0][simd_vec]` indexing. The match and replacements here are only different by the fact that we are now modifying the elemental kernel itself and not a skeleton.

V. Evaluation and Performance

In contrast to the OP2-Clang translator, OP2's current Python based translator only parses the application source so far as to identify OP2 API calls. However, no AST is created as a result, but simply the specifications and arguments in each of the `op_par_loop` calls are collected and stored in Python lists. When it comes to generating code, the full source of what is to be generated is produced, using the information gathered in these lists, for each of the parallelizations. No text replacements are done as in the OP2-Clang translator. However, as the invariant code for a given parallelization can be generated without change, only the specific changes for a given `op_par_loop` needs to be produced. Again, the code generation stage does not use an AST. As such, changing code within elemental kernels (as in the SIMD Vectorization case) is significantly difficult and cannot easily handle different ways a user might write their elemental kernels. All of the above makes the Python translator error prone and difficult to extend and maintain. In this section we present some results from evaluating the OP2-Clang translator on two OP2 applications by comparing code generated from it to the performance of the code generated through OP2's current Python based translator.

Both of the applications used in these tests have loops with indirections and indirect increments, various global reductions and use of global constants. The first, Airfoil, is a benchmark application, representative of large industrial CFD applications utilized by users of OP2. It is a non-linear 2D inviscid airfoil code that uses an unstructured grid and a finite-volume discretization to solve the 2D Euler equations using a scalar numerical dissipation [11]. The mesh used in our experiments consists over 2.8 million nodes cells and about 5.8 million edges. Airfoil consists of five computational loops and in the most computational intensive loop about 100 floating-point operations performed per mesh edge. The second application, Volna is a shallow water simulation capable of handling the complete life-cycle of a tsunami (generation, propagation and run-up along the coast) [12], [13]. The simulation algorithm works on unstructured triangular meshes and uses the finite volume method. For the experiments we used a mesh with about 2.4 million cells and about 3.5 million edges.

The generated code were compiled and executed on a single Intel Xeon CPU E5-1660 node (total of 8 cores) for OpenMP and SIMD vectorization (using Intel compilers suite 17.0.3.) and a single NVIDIA P100 GPU with CUDA 9.0. Table I shows the performance results and percentage difference of runtime compared to OP2's current Python-based translator. In all cases the performance difference is less than 2%. This figure was the same when comparing run-times of each kernel. These results, therefore gives an initial indication that identical code was generated by OP2-Clang. We are currently carrying out further tests to identify any differences in performance. As mentioned the absolute performance of production applications using OP2 has been published previously in [26], [45].

VI. Conclusions and Future Work

In this paper we introduced OP2-Clang, a source-to-source translator based on LibTooling, for OP2. OP2-Clang is capable of parsing a higher-level declarative programme written in OP2's C/C++ API and generate parallel code based on SIMD, OpenMP, CUDA and their combinations with MPI. The wide range of transformations required for generating code for each parallelization in OP2 and the variation in specific optimizations for each are significant, going well beyond what has been previously demonstrated with LibTooling. We presented the use of parallelization skeletons to reuse code and demonstrated the use of LibTooling's `ASTMathchers` and `Replacements` to modify a skeleton to generate the necessary parallel code. Multiple levels of refactoring using LibTooling's `RefactoringTool` enables to apply specific optimizations in a flexible, maintainable and extensible manner. Challenges in developing OP2-Clang were presented, discussing the generation of MPI, OpenMP, SIMD vectorized code and CUDA code for CPUs and GPUs. Performance from the OP2-Clang generated code showed near-identical performance to the code generated by OP2's current source-to-source translator (based on Python). We believe that the lessons learnt from OP2-Clang can be readily applied in developing similar source-to-source translators, particularly for DSLs.

The next stage of this work will add semantic checks over the generated code at the time of the translation to help developers debug their OP2 applications. At this point OP2-Clang will be ready to replace the existing translator in OP2. Longer term objectives of this research will look in to how information about the high-level OP2 application can be propagated down to the IR level. The open research question is whether we can propagate information that we know holds true given the OP2 abstraction to the LLVM optimizer. This will then enable LLVM to do optimizations that it would have not identified or attempted if the same application was developed as a general purpose programme using conventional C++. This we hope will enable us to enforce more aggressive optimizations, resulting in higher performance. Additionally we hope also to generalize the optimizations we discover while working with OP2-Clang to all C++ programs supported by Clang/LLVM. We also believe that this work can be extended for OP2's Fortran API, if a similar infrastructure is developed around the upcoming Flang [46] front end.

The full source of OP2-Clang is available as open source software at [10]. OP2, and the Airfoil application is available at [1]. Volna is available at [45]. The authors welcome new users and developers to these projects.

Acknowledgements

The authors would like to acknowledge the use of the University of Oxford Advanced Research Computing (ARC) facility in carrying out this work http://dx.doi.org/10.5281/zenodo.22558. István Reguly was supported by the János Bolyai Research Scholarship of the Hungarian Academy of Sciences. Project no. PD 124905 has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the PD_17 funding scheme. The research has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00013). OP2 was developed with funding from the UK Technology Strategy Board and Rolls-Royce Plc. through the Siloet project and the UK Engineering and Physical Sciences Research Council projects EP/I006079/1, EP/I00677X/1 on 'Multi-layered Abstractions for PDEs.

References

[1] "OP2 github repository." https://github.com/OP2/OP2-Common.

[2] G. Mudalige, M. Giles, I. Reguly, C. Bertolli, and P. Kelly, "Op2: An active library framework for solving unstructured mesh-based applications on multi-core and many-core architectures," in *Innovative Parallel Computing (InPar), 2012*, pp. 1–12, IEEE, 2012.

[3] D. J. Quinlan *et al.*, "Rose compiler project," 2012.

[4] P. Yang, F. Dong, D. Williams, J. Roerdink, B. Liu, A. Anvari-Moghaddam, G. Min, *et al.*, "Improving utility of gpu in accelerating industrial applications with user-centred automatic code translation," *IEEE Transactions on Industrial Informatics*, 2017.

[5] D. Williams, V. Codreanu, P. Yang, B. Liu, F. Dong, B. Yasar, B. Mahdian, A. Chiarini, X. Zhao, and J. B. Roerdink, "Evaluation of autoparallelization toolkits for commodity gpus," in *International Conference on Parallel Processing and Applied Mathematics*, pp. 447–457, Springer, 2013.

[6] "Libtooling," 2018.

[7] E. Bendersky, "Modern source-to-source transformation with clang and libtooling," 2014.

[8] J. Wu, A. Belevich, E. Bendersky, M. Heffernan, C. Leary, J. Pienaar, B. Roune, R. Springer, X. Weng, and R. Hundt, "gpucc: an open-source gpgpu compiler," in *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, pp. 105–116, ACM, 2016.

[9] M. Marangoni and T. Wischgoll, "Togpu: Automatic source transformation from c++ to cuda using clang/llvm," *Electronic Imaging*, vol. 2016, no. 1, pp. 1–9, 2016.

[10] "OP2-Clang github repository." https://github.com/bgd54/OP2-Clang.

[11] M. B. Giles, G. R. Mudalige, Z. Sharif, G. Markall, and P. H. Kelly, "Performance analysis and optimization of the op2 framework on many-core architectures," *The Computer Journal*, vol. 55, no. 2, pp. 168–180, 2011.

[12] D. Dutykh, R. Poncet, and F. Dias, "The volna code for the numerical modeling of tsunami waves: Generation, propagation and inundation," *European Journal of Mechanics-B/Fluids*, vol. 30, no. 6, pp. 598–615, 2011.

[13] I. Z. Reguly, D. Gopinathan, J. H. Beck, M. B. Giles, S. Guillas, and F. Dias, "The volna-op2 tsunami code (version 1.0)," *Geoscientific Model Development Discussions*, 2018.

[14] D. Unat, X. Cai, and S. B. Baden, "Mint: realizing cuda performance in 3d stencil methods with annotated c," in *Proceedings of the international conference on Supercomputing*, pp. 214–224, ACM, 2011.

[15] C. Bertolli, A. Betts, G. Mudalige, M. Giles, and P. Kelly, "Design and performance of the op2 library for unstructured mesh applications," in *European Conference on Parallel Processing*, pp. 191–200, Springer, 2011.

[16] S.-Z. Ueng, M. Lathara, S. S. Baghsorkhi, and W. H. Wen-mei, "Cuda-lite: Reducing gpu programming complexity," in *International Workshop on Languages and Compilers for Parallel Computing*, pp. 1–15, Springer, 2008.

[17] S. Lee, S.-J. Min, and R. Eigenmann, "Openmp to gpgpu: a compiler framework for automatic translation and optimization," *ACM Sigplan Notices*, vol. 44, no. 4, pp. 101–110, 2009.

[18] S.-I. Lee, T. A. Johnson, and R. Eigenmann, "Cetus–an extensible compiler infrastructure for source-to-source transformation," in *International Workshop on Languages and Compilers for Parallel Computing*, pp. 539–553, Springer, 2003.

[19] T. D. Han and T. S. Abdelrahman, "hi cuda: a high-level directive-based language for gpu programming," in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pp. 52–61, ACM, 2009.

[20] O. Krzikalla, K. Feldhoff, R. Müller-Pfefferkorn, and W. E. Nagel, "Scout: A source-to-source transformator for simd-optimizations," in *Proceedings of the 2011 International Conference on Parallel Processing - Volume 2*, Euro-Par'11, (Berlin, Heidelberg), pp. 137–145, Springer-Verlag, 2012.

[21] I. Z. Reguly, G. R. Mudalige, M. B. Giles, D. Curran, and S. McIntosh-Smith, "The ops domain specific abstraction for multi-block structured grid computations," in *Proceedings of the 2014 Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, WOLFHPC '14, pp. 58–67, IEEE Computer Society, 2014.

[22] F. Rathgeber, D. A. Ham, L. Mitchell, M. Lange, F. Luporini, A. T. T. Mcrae, G.-T. Bercea, G. R. Markall, and P. H. J. Kelly, "Firedrake: Automating the finite element method by composing abstractions," *ACM Trans. Math. Softw.*, vol. 43, pp. 24:1–24:27, Dec. 2016.

[23] C. T. Jacobs, S. P. Jammy, and N. D. Sandham, "OpenSBLI: A framework for the automated derivation and parallel execution of finite difference solvers on a range of computer architectures," *Journal of Computational Science*, vol. 18, pp. 12–23, 2017.

[24] M. Lange, N. Kukreja, M. Louboutin, F. Luporini, F. Vieira, V. Pandolfo, P. Velesko, P. Kazakas, and G. Gorman, "Devito: Towards a generic finite difference dsl using symbolic python," in *Proceedings of the 6th Workshop on Python for High-Performance and Scientific Computing*, PyHPC '16, (Piscataway, NJ, USA), pp. 67–75, IEEE Press, 2016.

[25] N. Jacobsen, "Llvm supported source-to-source translation-translation from annotated c/c++ to cuda c/c++," Master's thesis, 2016.

[26] I. Z. Reguly, G. R. Mudalige, C. Bertolli, M. B. Giles, A. Betts, P. H. Kelly, and D. Radford, "Acceleration of a full-scale industrial cfd application with op2," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 5, pp. 1265–1278, 2016.

[27] K. B. Ølgaard, A. Logg, and G. N. Wells, "Automated Code Generation for Discontinuous Galerkin Methods," *CoRR*, vol. abs/1104.0628, 2011.

[28] F. Rathgeber, D. A. Ham, L. Mitchell, M. Lange, F. Luporini, A. T. T. McRae, G.-T. Bercea, G. R. Markall, and P. H. J. Kelly, "Firedrake: Automating the Finite Element Method by Composing Abstractions," *ACM Transactions on Mathematical Software*, 2017.

[29] C. T. Jacobs and M. D. Piggott, "Firedrake-Fluids v0.1: numerical modelling of shallow water flows using an automated solution framework," *Geoscientific Model Development*, vol. 8, no. 3, pp. 533–547, 2015.

[30] P. Vincent, F. Witherden, B. Vermeire, J. S. Park, and A. Iyer, "Towards green aviation with python at petascale," in *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–11, Nov 2016.

[31] I. Z. Reguly, G. R. Mudalige, M. B. Giles, D. Curran, and S. McIntosh-Smith, "The ops domain specific abstraction for multi-block structured grid computations," in *2014 Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, pp. 58–67, Nov 2014.

[32] M. Lange, N. Kukreja, M. Louboutin, F. Luporini, F. Vieira, V. Pandolfo, P. Velesko, P. Kazakas, and G. Gorman, "Devito: Towards a generic finite difference dsl using symbolic python," pp. 67–75, IEEE, 2016.

[33] T. Gysi, C. Osuna, O. Fuhrer, M. Bianco, and T. C. Schulthess, "Stella: A domain-specific tool for structured grid methods in weather and climate models," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, (New York, NY, USA), pp. 41:1–41:12, ACM, 2015.

[34] "PSyclone Project - GitHub Repository," 2018. https://github.com/stfc/PSyclone.

[35] H. Carter Edwards, C. R. Trott, and D. Sunderland, "Kokkos," *J. Parallel Distrib. Comput.*, vol. 74, pp. 3202–3216, Dec 2014.

[36] R. D. Hornung and J. A. Keasler, "The RAJA portability layer: Overview and status," tech. rep., Lawrence Livermore National Lab. (LLNL), 9 2014.

[37] M. B. Giles, G. R. Mudalige, B. Spencer, C. Bertolli, and I. Reguly, "Designing op2 for gpu architectures," *Journal of Parallel and Distributed Computing*, vol. 73, no. 11, pp. 1451–1460, 2013.

[38] G. Mudalige, M. Giles, J. Thiyagalingam, I. Reguly, C. Bertolli, P. Kelly, and A. Trefethen, "Design and initial performance of a high-level unstructured mesh framework on heterogeneous parallel systems," *Parallel Computing*, vol. 39, no. 11, pp. 669 – 692, 2013.

[39] "Matching the clang ast." https://clang.llvm.org/docs/LibASTMatchers.html, 2018.

[40] "Refactoringtool class reference."

[41] E. Bendersky, "Ast matchers and clang refactoring tools," 2014.

[42] "Replacements class reference," 2018.

[43] A. A. Sulyok, G. D. Balogh, I. Z. Reguly, and G. R. Mudalige, "Improving locality of unstructured mesh algorithms on gpus," *CoRR*, vol. abs/1802.03749, 2018.

[44] G. R. Mudalige, I. Z. Reguly, and M. B. Giles, "Auto-vectorizing a large-scale production unstructured-mesh cfd application," in *Proceedings of the 3rd Workshop on Programming Models for SIMD/Vector Processing*, WPMVP '16, (New York, NY, USA), pp. 5:1–5:8, ACM, 2016.

[45] "OP2-Volna github repository." https://github.com/reguly/volna.

[46] "Flang: a fortran compiler targeting llvm.," 2018.